TDB-ACC-NO: NN76053971

DISCLOSURE TITLE: User Configurable Local Storage Registers. May 1976.

PUBLICATION-DATA: IBM Technical Disclosure Bulletin, May 1976, US

VOLUME NUMBER: 18

ISSUE NUMBER: 12

PAGE NUMBER: 3971 - 3975

PUBLICATION-DATE: May 1, 1976 (19760501)

CROSS REFERENCE: 0018-8689-18-12-3971

DISCLOSURE TEXT:

5p. A mechanism is described for use with a data processor wherein different types of registers such as general-purpose registers, index registers, floating-point registers and the like are located in a local storage array. The mechanism enables the programmer or user to customize or configure the register space in local storage, so as to provide the optimum arrangement of registers in terms of numbers and types to best suit his particular application.

- In processors currently on the market in which user addressable registers are **implemented** in local storage, optimum utilization of the local storage resource is not always completely possible in those cases where more than one type of user addressable register exists in the architecture. This is because the partitioning of the local storage array into sets of registers of particular types may not be appropriate for the application to be processed. For example, a processor may have special registers for floating-point arithmetic whose array space could be far better utilized as additional index registers by an application which performs no floating-point operations.

- In order to explain the basic concepts of the technique described herein, a local storage array having the dimensions shown in the drawing will be used by way of example to explain one possible implementation of the new technique. This array has 64 addressable locations and each location has a width of 128 bits. Addressing any given location will cause a parallel readout of all 128 bits at such location by way of a 128-conductor array output data bus. For convenience, the 128 bits at each location are called a full array word. The registers being considered herein are to be located in the high-address half (locations 32-63) of the local storage array.

- For sake of example, it is assumed that the following five

types of registers can be located in the register space of the local storage array:

32-bit index registers.
64-bit binary/logical accumulators.
64-bit hexadecimal floating-point registers.
128-bit decimal floating-point registers.
128-bit addressing registers.

- Taking 128 bits as being a full array word, the 32-bit registers will be called quarterword registers, the 64-bit registers will be called halfword registers and the 128-bit registers will be called fullword registers. Four typical quarterword registers are indicated at A, B, C and D in the drawing. Two typical halfword registers are indicated at E and F and a typical fullword register is indicated at G. The choice of the number of each type and of their location in the register space is under the control of the programmer, except that the proper boundary alignment conditions must be observed. In other words, any given halfword register must be either left aligned (position E) or right aligned (position F).
- In a similar vein, any given quarterword register must be aligned in one of the quarterword positions represented by positions A-D. Also, a fullword register must be located entirely at the same storage location in the manner indicated at G. Apart from these boundary alignments, the programmer has complete freedom as to the number and location of the different register types he may wish to use.
- It is further assumed that if a given operand of a given machine instruction is a register, then it is a particular register type and may only reference a register of that type. In other words, the register type or types for any given machine instruction are predesigned and cannot be varied.
- In order to be able to pack the registers tightly into the available local storage space, it is required that the busing between the processor and the local storage array be able to handle at least 7 types of fetches and stores, namely: fullword (128 bit bits), left and right aligned halfwords (64 bits), and first, second, third and fourth aligned quarterwords (32 bits). Note that the choice of lengths as powers of two simplifies both the busing and packing problems considerably. Other types of accesses may, of course, be provided as required to handle the particular coding of certain register types such as, for example, floating-point exponents.
- Since the herein described mechanism is able to use any quarterword portion of the register space of the array as a 32-bit index register, the mechanism must be able to distinguish up to 128 different registers. This calls for all register designation fields or register "tags" in the machine instructions to be at least 7 bits wide. That is, every operand of every machine instruction which is a register is coded in the instruction format as a 7-bit field, thus providing tag values ranging from zero to 127.
- The machine instruction tag value describing an index register

(32 bits) can be any of the above values (0-127), and index registers may appear in any aligned quarterword of the local storage register space. The tag value describing either type of 64-bit register is constrained to be an even number (xxxxxx0) and these types of registers are constrained to be either right or left aligned in an array location. Similarly, the tag values describing either type of 128-bit register are constrained to be a multiple of four (xxxxx00) and these registers are constrained to occupy a complete array location. Thus, there may be at most 128 32-bit registers or 64 64-bit registers or 32 128-bit registers allocated in the available 4096 bits of register space.

- When the different size registers are allocated in combinations, they are required not to overlap. This means that certain register tag values may never be used. For example, if register "4" is a 128-bit register, there will be no registers referenced as "5", "6", or "7" because the space associated with these tag values is "inside" register "4".
- When the processor is to interpret a machine instruction containing a register reference, it must relate the coded tag value with an area in the local storage register space. This involves determining the array word (location), the offset within the word and the width of the register. The array word is selected by concatenating a one bit at the high-order end of the high-order five bits of the register tag value, to take into account the fact that the register space is in the high address half of the array.
- The resulting 6-bit word address is sent to the local storage address decoder in the normal manner. The offset is obtained directly by taking the last or low-order two bits of the register tag value. These offset bits are sent to the data bus selector to control the busing of the selected quarterword, halfword or fullword.
- Finally, the width of the register is obtained from the control register decodes by noting what operation **code** (op **code**) is present and which operand of that op **code** is being dealt with. It is thus possible for the addressing mechanism of the local storage array to be preset by instruction cycles to identify the register operands, without having to consider separately the semantics of the instructions.
- The programming procedures and mechanisms to be employed in establishing a desired register configuration in the local storage array will now be described. The description will be given from the viewpoint of a programmer working at the assembler language level.
- The programmer, knowing what types of registers are available and how much total space his registers may use, decides on the configuration that would be best for the **program** he is about to write. In most cases, there will also be alignment considerations due to the capabilities of the hardware.
- The desired mapping of register types onto the local storage array hardware is recorded in the source **program** in the form of DECLARE statements. Each DECLARE statement associates a particular

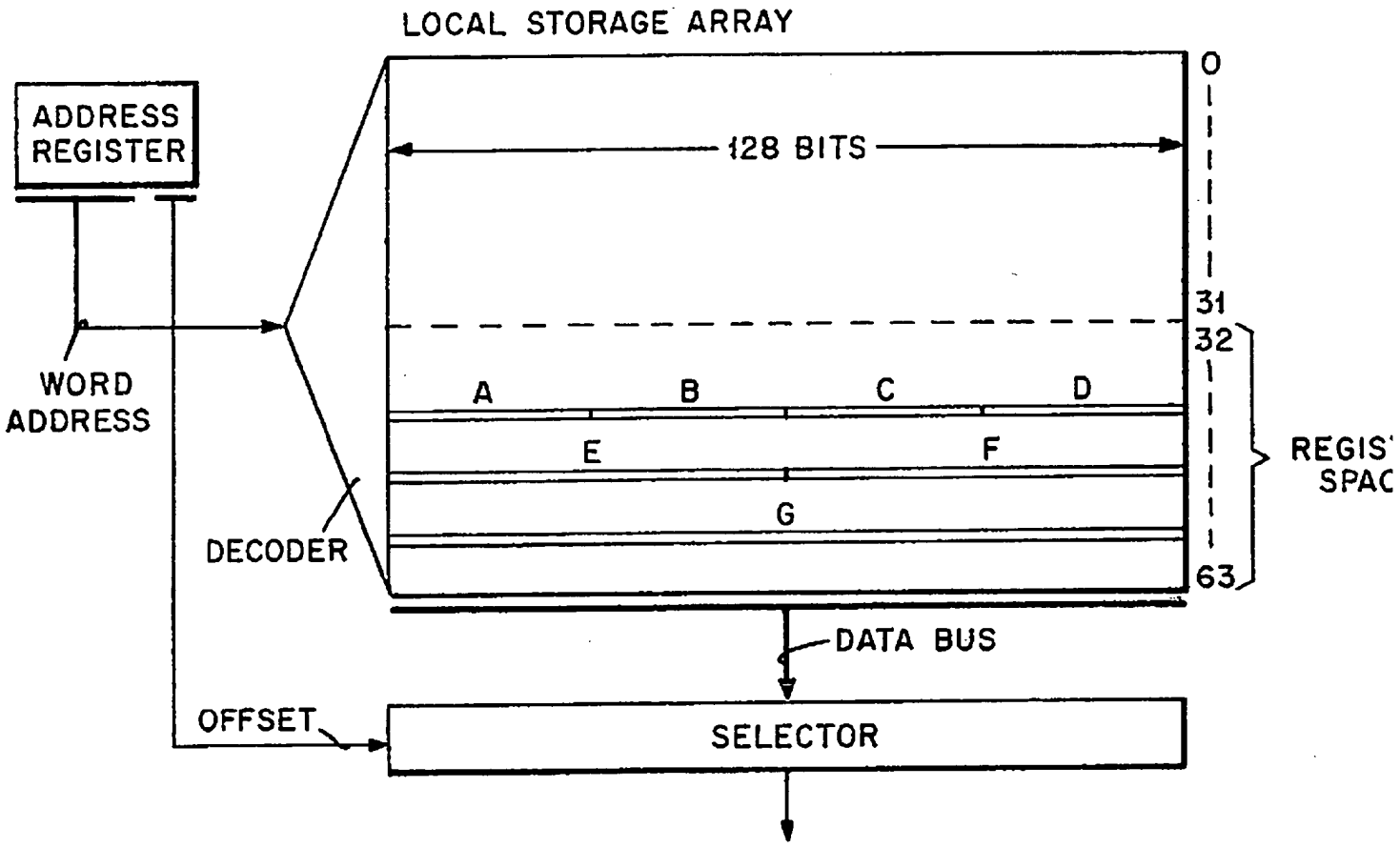
register tag value with a particular register type. For example, the programmer might write the following instruction:
DCL ADR1 4

This is a DECLARE (DCL) statement saying that register tag value "4" is to be assigned to addressing register number 1. The register length is implied by its type. In this case, the register type is an addressing register and is thus known to be four quarterwords (128 hits) in length. The tag value "4" says that this register's starting address is 0000100 in terms of the 7-bit tag **code**.

- The assembler **program** translates the **program** from its source representation to the internal machine language representation. Unlike conventional practice, however, the DECLARE statements describing registers are also included in the machine language representation. It is immaterial to the mechanism whether register usage has been checked for correctness at this point.
- The machine language **program** is then validated and made executable. In a preferred implementation of this technique being described, this step involves the encapsulation of the **program** into a **program** module that is both unmodifiable and uncounterfeitable by the programmer. In any event, at this point in the process, the **program** is validated by **checking for conformance** to syntax rules including those relating to the register DECLARE statements. The illegal register conditions which must be detected at this time are those which will not be checked at execution time. Two user errors in particular are relevant to the proposed mechanism, namely, the declaration of registers with illegal alignments and the inappropriate, illegal or impossible usage of register tags in the instructions.
- When the **program** is executed, the actions taken by the hardware and microcode are identical with current practice. The values of the register tags are used to locate operands in the local storage register space. The length and encoding of the operands is determined from the op **code** and the position of the operand in the machine instruction being considered. The machine operations to be performed are determined from the op **code**.
- It is seen from the foregoing that considerable additional flexibility has been given to the programmer to optimize local storage resource usage. At the same time, almost no execution time overhead is incurred thereby. In this regard, it is noted that some additional processing time is required to perform the desired checking for improper usage of register tags. This, however, is not done at execution time. In other words, if integrity of register type is required, some preprocessing of executable **programs** is required. This requires some additional capabilities in the machine and some additional processing time, but this is on a per-**program** compiled basis and not on a perinstruction-executed basis.
- The basic concept of allowing the programmer to define the register configuration to be used in local storage is not limited to

the particular relationship between local storage dimensions and register widths shown in the drawing. For example, the basic concept is equally applicable to the case where the width of the local storage array is the same as the width of the smallest size register that can be specified.

- In summary, the foregoing technique enables the programmer to customize the register population of the data processing machine in a manner similar to DECLARE-ing variables to a compiler. Little or no execution time degradation is experienced and full integrity of register types is achievable.


LOCAL STORAGE ARRAY
ADDRESS REGISTER
128 BITS
0
31
32
63
WORD ADDRESS
A
B
C
D
E
F
G
DECODER
DATA BUS
OFFSET
SELECTOR